A communication framework for distributed access control in microkernel-based systems

Mohammad Hamad, Johannes Schlatow, Vassilis Prevelakis and Rolf Ernst
Institute of Computer and Network Engineering, TU Braunschweig
{mhamad,schlatow,prevelakis,ernst}@ida.ing.tu-bs.de

***Abstract*—Microkernel-based architectures have gained an increasing interest and relevance for embedded systems. These can not only provide real-time guarantees but also offer strong security properties which become increasingly significant in certain application domains such as automotive systems. Nevertheless, the functionality of those complex systems often needs to be distributed across a network of control units for various reasons (e.g. physical location, scalability, separation). Although microkernels have been commercially established, distributed systems like these have not been a major focus. This is basically originated by the fact that – in the microkernel world – policy, device drivers and protocol stacks are userspace concerns and rather left to be solved by the particular application domain. Following the principle of least privilege, we therefore developed a distributed access-control framework for all network-based communication in microkernel-based systems that can be generically deployed. Our design not only enforces security properties such as integrity but is also scalable without adding too much overhead in terms of run time or code.**

I. Introduction

Nowadays embedded systems are ubiquitous, i.e. in most of the electronic devices in our life; from simple devices such as microwaves to sophisticated ones such as cars. The latter is a driving motor when it comes to safety concerns of complex embedded systems, which is typically approached by a deliberate system design that uses separation and "safety nets". A contemporary car contains from 70 to 100 microcontroller-based computers [1], known as electronic control units (ECUs). These ECUs control many functions within the car, ranging from the mundane such as controlling courtesy lights to the highly critical such as engine control. These ECUs are distributed around the vehicle and interconnected using different bus systems such as CAN, MOST or FlexRay in a rather static setup. The need of exchanging bigger and more expressive messages is pushing towards using Internet Protocol (IP) standards for both on-board and vehicle-to-X communications [2], [3], which is also driven by the desire of better modifiability and updateability in the automotive domain. However, one main reason that connected ECUs are becoming increasingly vulnerable is the use of unprotected wireless and wired communication [4].

Increasing the flexibility of vehicular (software) platforms while not neglecting the safety and security therefore is a major challenge. Moreover, in contrast to traditional computing systems, embedded systems not only come with limited resources concerning memory and CPU power but also have slightly different demands on the system's security.

Modern sedans run a huge number of applications (functions) with millions of lines of code (LOC) [1]. These applications come from several vendors with various levels of code quality. Safety-relevant functions, such as anti-lock braking systems, are typically well-engineered and heavily tested, while others, such as the entertainment systems, could be implemented with security and reliability not as prime factor. The uncontrolled interference within shared buses between applications with a mixed level of safety, security and criticality may create vulnerabilities [5]. Compromising uncritical components by an adversary could be sufficient to control critical components across the entire car, which must be dealt with by appropriate protection mechanisms. Using microkernels could be the first step towards providing a secure environment for such systems [6], which will benefit from the small amount of privileged code, the minimal trusted computing base (TCB), and the memory protection between the different components. However, providing a comprehensive framework for controlling the communication between the various platform subsystems is a crucial complementary task to the microkernel's security services.

We have created a distributed access-control framework which allows only authorized components to interact with each other inside the vehicle and with external entities. Our framework ensures the ability to define the type of security provided for each communication link (e.g. integrity, confidentiality), and other connection properties (e.g. priority). We defined a secure communication policy, which determines all permitted paths between different components, centrally and gradually. Later on, the policy was enforced by each ECU on the vehicle isolatedly (i.e. without any need for additional interactions).

The rest of the paper is organized as follows. Section II describes the communication framework and the objectives of our work. It also explains the security approach of local and remote communication between the different components. In Section III, we describe the main parts of the communication module and its implementation. Finally, related work is presented in Section IV before we evaluate our implementation in Section V and discuss our findings in Section VI.

II. Communication Framework

Fig. 1 depicts an exemplary distributed system and the different scenarios of inter-component communication. In our idealistic world, each ECU is running a microkernel-based operating system that hosts multiple (interacting) software

https://doi.org/10.24355/dbbs.084-201806251501-3

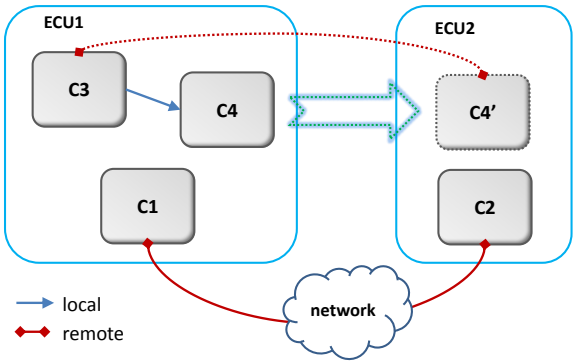


Fig. 1. Communication scenarios in distributed automotive systems

components. The ECUs are then inter-connected by a network. Here, we identify two types of inter-component communication: *Local* communication, which refers to the communication between two components on the same ECU (e.g. C3 and C4), and *remote* communication, which denotes the communication between components on two different ECUs (e.g. C1 and C2).

Our goal is to build a framework that ensures secure communication between the components in this scenario while still maintaining a high flexibility. For this purpose, we first state our objectives before we discuss local communication mechanisms w.r.t. their application in our scenario and finally derive our networking architecture approach.

A. (Security) objectives

When it comes to designing a communication framework for distributed systems in critical application domains such as automotive systems, we identified the following objectives:

1) Fine-grained access control: Our primary goal is to control who should talk to whom in an efficient manner without the need for static access-control mechanisms. Components should only communicate with other components who are specified by the policy. One main advantage of this is that even a compromised component will only be able to interact with authorized components and will be unable to attack other components indiscriminately.

2) Secure communication: Providing security services for authorized connections is a fundamental requirement. The required security services are varying from one application to another. However, since most security issues in vehicle communications are related to the lack of authentication mechanisms, providing integrity and mutual authentication is necessary to prevent unauthorized parties from sending false data or injecting them in established connections.

3) Composability and migratability: In a component-based system, the desired functionality is integrated by composing several interacting components. In a distributed scenario, we also gain the freedom of choice where to execute each component. Fig. 1 illustrates this on component C4 which could alternatively be executed on ECU2 but then requires a remote communication mechanism to C3. Hence, composability and migratability are important values whose lack would quickly restrict the design space. Note that we consider migration in terms of a (partial) system reconfiguration that must undergo several admission tests before being applied.

4) Minimum (application-specific) TCB: A common goal when building secure systems is the minimization of the TCB, i.e. the subset of hardware and software that must be relied upon. Microkernels already do their share w.r.t. minimizing the TCB. Yet, in userspace, a flawed design can easily bloat the TCB, e.g. by adding a middleware for all applications. Instead, each application should only rely upon a minimum set of software components with minimum complexity and therefore have its specific TCB [7].

5) Legacy application support: Another concern is the ability to integrate legacy applications into a component-based system. Note that legacy APIs might need to be monitored and restricted so that they do not conflict with the above objectives. In the scope of this work, we demonstrate the feasibility of this by providing a socket API to conventional network applications.

B. From IPC towards networked communication

Any microkernel architecture provides strong isolation of application components in order to minimize the TCB. Therefore, any communication between isolated components, i.e. inter-process communication (IPC), needs to be mediated by the kernel. On the one hand, this introduces additional overhead which was historically one of the main drawbacks of the microkernel approach but was weakened by the optimization of (synchronous) IPC mechanisms and the evolution of microkernels [8], [9]. On the other hand, this has the benefit of making any communication explicit. This property was further strengthened by introducing capability-based access control that enables a fine-grained and unforgeable control of a component's communication channels. As a result, today's microkernel architectures allow us to apply the principle of least privilege and enforcing security policies when integrating application components from different, potentially untrusted, parties [10]. In summary, all these properties helped establishing microkernels as sophisticated, and also commercialized [11]–[14], implementation vehicles for critical application domains.

When it comes to more dynamic scenarios like distributed systems, a service-oriented approach is commonly taken to equip the system with the required flexibility. Typically, a communication middleware then takes care of routing the messages to the communication partner that registered under a certain service name thereby deploying a communication mechanism (API) that is agnostic of the actual communication partners and their location. Yet a major drawback of such a middleware is that enforcing security policies and providing isolation (e.g. local namespaces) while not adding too much overhead is a non-trivial obligation. This approach clearly trades ease-of-use against simplicity and efficiency.

We therefore believe that the strong architectural guarantees already provided by microkernels can and should be utilized in such scenarios. That means local communication shall still benefit from the existing efficient and secure implementations

https://doi.org/10.24355/dbbs.084-201806251501-3

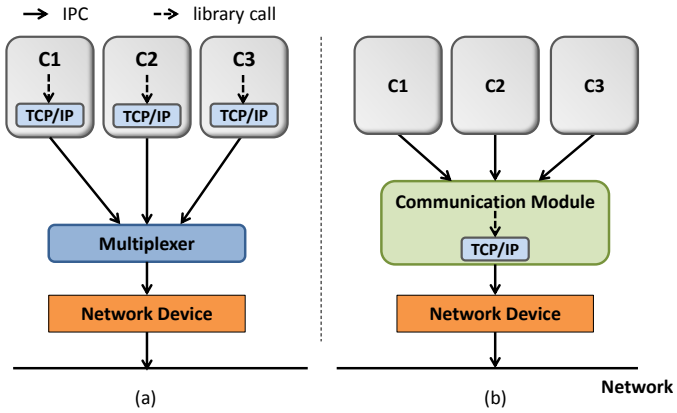


Fig. 2. Architecture with multiple TCP/IP stacks and a shared multiplexer (a) compared to a shared communication module with an integrated TCP/IP stack (b).

while we transparently transform between the local and remote communication mechanisms where necessary. The challenge here is to provide similar guarantees for remote communication, i.e. the fine-grained access control and integrity of remote communication channels. Yet there is a mismatch between the fine-grained access control for local IPC and the socket API typically used for network applications that gives full access to any attached network. We therefore need to provide the infrastructure with which we can control network accesses in order to establish unforgeable network communication between application components as we are used to on when using local IPC.

C. From user-level networking towards a distributed firewall

Microkernel philosophy is based on moving all policy, including device drivers and protocol stacks, from the kernel to the userspace. Therefore, implementing the network stack in the userspace was a hot topic for many years; it was proposed for different motivations, including increasing the performance and flexibility of the network layer [15].

Providing maximum isolation between different applications, a straight-forward approach of executing several network applications on a microkernel consists in using dedicated network stacks for each application (cf. Fig. 2a). Here, the low-level Network Interface Controller (NIC) must be multiplexed/virtualized, e.g. by a network bridge. As a result each application is linked to its network-stack library and requires its own MAC and IP addresses. The drawback of this approach is that unless some sort of packet filtering is deployed, each application also gets full access to the shared network, which contradicts our objective of a fine-grained access control. More precisely, this approach is susceptible to the following communication threats:

a) Spoofing: An application, which has a full access to the network stack, can emit a frame with fake IP or MAC addresses. Such an application may imitate other applications, eavesdrop on their communication, or collect relevant information about the platform. It could also change the transmitted data and inject false values.

```
Authorizer:    Integrator_public_key
Licensees:     Platform_public_key
Conditions:    (Vendor_id ==“ACME_INSTRUMENTS“
                 && Src_device_name  == “headlight_control“
                 && Dst_device_name == “ambient_light_sensor“
                 &&  Src_device_type  == CONTROL_PLATFORM
                 &&  Dst_device_type  == LIGHT_SENSOR
                 && Security_level  >= SL_INTEGRITY  && Priority_level  ==  HIGH
                 && Bitrate_limit == X Kbitd/sec )  ->  “ALLOW“
Signature:     Integrator signature
```

Fig. 3. An example of KeyNote credential which enables an ambient light sensor to communicate with headlight control. The credential ensures integrity of the communication line with high priority.

b) Denial of service (DoS): One of the primary results of the IP spoofing can be a DoS attack. I.e. a malicious application could spoof a target service's IP address and send many packets to different receivers. All responses to the spoofed packets will be directed to the services IP, which will be flooded. Sometimes, an attack cannot cause a disruption of the service, but can cause a degradation of its quality (e.g. by increasing its response time). The DoS could lead to serious issues if that service is responsible for the users' safety.

In order to combat these threats, adequate access-control mechanisms should be implemented to control the interaction between different applications and to prevent unauthorized parties from processing foreign data. However, packet filtering is more network-centric and typically too abstract for fine-grained application-level access control. Moreover, there is a consistency challenge when it comes to updating static filtering rules in a distributed system in case of a dynamically changing environment.

Integrating other traditional network protection methods such as firewalls in vehicle communication networks was shown to be inadequate too [16], especially if the integrator keeps the assumption that all insider nodes are trusted. Moreover, using a single ECU as a firewall to control the whole communications within the vehicle is also not an optimal solution. Such an ECU will create congestion, become a single point of failure and jeopardize the scalability.

Hence, adopting the distributed firewall technique [17] seems to be a favorable solution in order to remove any performance bottleneck. We applied this method by providing a single communication module for each ECU as shown in Fig. 2b. This module is playing the role of a firewall by controlling all incoming and outgoing communications on a single ECU and by enforcing the security policy locally. The security policy is managed centrally and then distributed to all ECUs. Note that the communication module implements a shared network stack and multiplexes the network device. It is therefore a potentially complex component that might compromise the isolation of the application components. We believe, however, that this design choice can actually simplify the policy enforcement and multiplexing task in contrast to solutions that implement these on other layers of abstraction.

In our previous work [18], we presented a mechanism to integrate the evaluation of communication policy into the

https://doi.org/10.24355/dbbs.084-201806251501-3

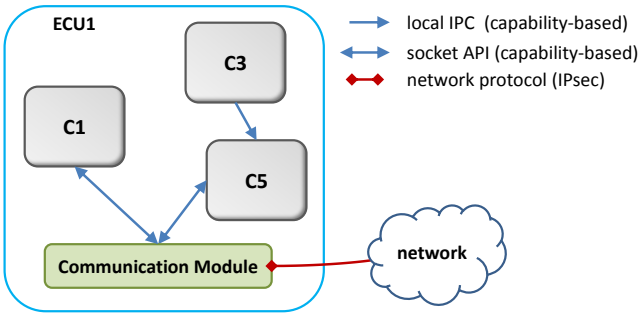


Fig. 4. Architecture with a shared communication module on each ECU

components' development flow; from the design process until the final integration stage of the component with the platform. Such integration will ensure that the defined policy will fulfill all the operational component requirements. Each component is identified by its own credential, which gives it the ability to communicate with other components regardless its topological location in the network.

We used the KeyNote policy definition language [19] to formulate the communication policy as shown in Fig. 3. The application-independent design of KeyNote allows for the support of a variety of different applications. KeyNote furthermore enables the delegation of the policy by allowing principals to delegate authorization to other principals (e.g. in Fig. 3 where the integrator delegates rights to the platform). Consequently, the delegation capability allows to decentralize the administration of policies.

III. Implementation of the Communication Module

We implemented our communication module for the Genode OS Framework as distinct userspace server which provides and monitors network accesses. Hence any network application acts as a client that connects to this server using a socket-like API. Fig. 4 illustrates the different communication scenarios (cf. Fig. 1): While C1 represents a network application component directly using the socket API, C3's local IPC is translated by C5 and the communication module into a network communication. An important detail here is the capability-based access control used to manage the access to the communication module. As a system integrator, we can thus perfectly control the local inter-component communication and thus guarantee that no application component has direct access to the network interface. Moreover, the communication module is able to distinguish its clients by their capabilities and can therefore select and enforce different (pre-defined) policies for the network communication. In this way, all network accesses are securely mediated by the communication module. Note that this is based on the assumption that capabilities cannot be arbitrarily delegated between application components.

The communication module is composed of four cooperating submodules as depicted in Fig. 5. The pseudo-socket layer plays a central role by providing a suitable interface to the applications as well as by coordinating the other submodules. In the remainder of this section, we elaborate on these submodules in more detail:

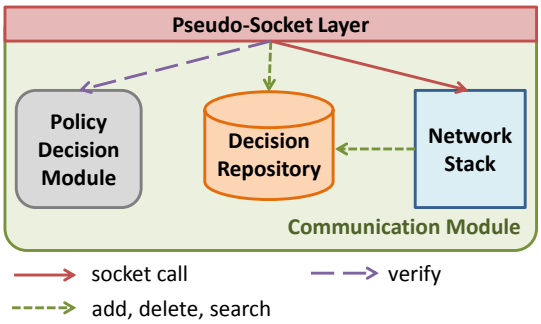


Fig. 5. Architecture of the communication module

a) Pseudo-socket layer: A lot of conventional (legacy) applications use a socket-like API (as in the standard C library) to access the network stack. Similarly, the pseudo-socket layer represents the interface which the applications use to interact with the communication module. In contrast to the conventional function/library calls, the pseudo-socket layer uses local IPC and must therefore take care of the memory management between the different address spaces of the communication module and its clients before a call can be handed over to the actual network stack. More precisely, we used shared memory to transfer the data between the address space of the application and the communication module to avoid imposing extra overhead by copying data multiple times. All this is transparently taken care of by this layer, so that the clients can still use the typical socket API functions. Legacy applications can be supported by linking against a slightly modified version of the standard C library that forwards the socket API calls to the communication module. Additionally, this layer checks the parameter validity, invokes the policy decision component with the relevant information related to the connection, and reacts to the received decision. If an affirmative decision (i.e. allow) is received, a new rule will be added to the repository. This rule contains many runtime specified selectors such as IP addresses and port numbers of the two ends of the connection. It also includes the required security level (i.e. integrity or confidentiality), the maximum allowed bit rate, and the priority level of the connection. Associating this rule with the opened socket gives us the ability to enforce this rule in two different layers. The first one is at the socket layer when an application uses the socket to send or receive data while the second one is placed at the IP layer whenever a new packet is received. The rule will be removed from the repository as soon as the socket is closed.

b) Policy Decision Module: This submodule is responsible for monitoring, i.e. granting or denying, the main requests of an application such as initiating a connection, sending, or receiving data. In order to make a decision, it determines whether a proposed request is consistent with the local policy and whether the conditions specified in the credentials were met. For this purpose, we use the KeyNote library.

c) Network Stack: As mentioned before, the network stack was integrated into the communication module to provide the basic network access. In our implementation, we

https://doi.org/10.24355/dbbs.084-201806251501-3

used the lightweight TCP/IP stack (lwIP). In addition, we integrated embedded IPsec [20] into this network stack (as proposed in [21]) in order to provide basic security services (e.g. integrity, confidentiality) to the clients. Furthermore, we consider implementing traffic monitoring in a later phase of our work to keep tracking of the bit rate of a connection in order to prevent DoS attacks, like proposed in [22].

d) Decision Repository: Providing a repository for saving the policy decisions is an essential technique in our design to spare the run-time costs of the request evaluation. By doing this, the evaluation only occurs when an application initiates a connection (i.e. accept() and connect() for TCP-based communication). For this purpose, the decision repository stores the decided rules for any opened connection.

IV. Related Work

Many authors have addressed deficiencies of vehicle communication and the need for a mechanism to control the interaction between the components within the vehicle and between the vehicle and the outside world [23]. However, only a few proposals have appeared to provide such a mechanism.

Based on legacy network solutions, Chutorash [16] proposes an approach for using a firewall to control the interaction between applications on the one side and vehicle bus and vehicle components on the other side. His approach was restricted to monitor the interaction between HMI systems and other vehicle components, which ignored controlling the interaction between the vehicle's components. We extended this approach by using a firewall for each ECU in order to build a distributed firewall that is concerned about all communication inside the vehicle.

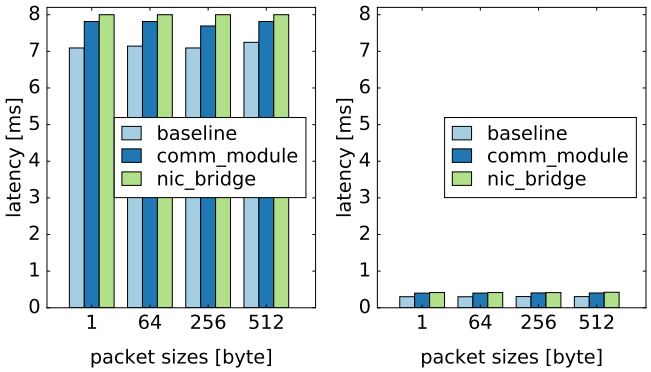
Concerning multiple network stacks, a userspace port switch was proposed in [24] that controls interconnecting independent network applications which run together. Swarm assigns the same IP, MAC address to all different stacks and uses port number to distinguish them. Yet using port numbers to control the communication flows is not sufficient when the applications use dynamic port assignment.

The Genode OS Framework [25] proposed the use of a NIC bridge which implements the Proxy-ARP protocol [26] to multiplex and monitor the communications of different applications that run on the same host. Neither solutions use filtering mechanisms which identify the application properly.

QNX Neutrino RTOS is a commercial microkernel-based real-time operating system that uses a networking architecture [27] very similar to what we propose. The dominant (local) IPC mechanism is synchronous message passing. Network communication is enabled via a socket API by a central network manager which implements device drivers and the network stack. In addition, the so-called Qnet protocol transparently extends the message-passing paradigm over a distributed system [28]. However, as Qnet is designed to be deployed for a group of trusted machines, it does not perform any authentication. Moreover, policy enforcement is only done in terms of packet filtering.

TABLE I
Communication module code size

Part	SLOC
Pseudo-socket layer	500
Policy Decision Module interface	300
IPsec extension of the Network Stack	2000
Decision Repository	600



(a) Genode on Raspberry Pi (b) Genode on Linux

Fig. 6. Average round-trip latency results for our two test platforms.

V. Evaluation

We evaluated our implementation of the communication module w.r.t. its overhead in terms of source lines of code (SLOC) and latency. Table I summarizes the SLOC values that have been acquired by the *cloc* tool. Note that we only evaluated the part of the policy decision module which interfaces the unmodified KeyNote library. It is also worth mentioning, that by our approaches saves about 750 SLOC by superseding the multiplexing component (i.e. nic_bridge).

Regarding the latency, we used the TCP_RR test of the netperf tool in order to measure the average round-trip latency. As a matter of course, the communication module must perform worse than a scenario where a single application directly accesses the network device. We therefore compared our approach against the scenario illustrated in Fig. 2a. More precisely, the multiplexer we used is the nic_bridge of the Genode OS Framework that implements the Proxy-ARP protocol. Hence the scenarios we compared both include additional copying and context switching caused by the nic_bridge and the communication module. The netserver component was executed on the system under test while the netperf binary was run from a standard linux machine. In particular, we added the parameters `-i 10,3 -I 99,5` in order to perform multiple iterations and achieve a confidence level of 99 %.

The average round-trip latency results are shown for different package sizes and two different platforms in Fig. 6. As a baseline, we also included the results for a setup in which the netserver directly accesses the network device. One of the test platforms was running Genode on a Raspberry Pi whereas the other platform was running Genode directly on the same linux machine as the netperf binary. Note that we used the latter to

https://doi.org/10.24355/dbbs.084-201806251501-3

bypass any physical network devices and drivers as it only utilizes rather simple virtual network interfaces. Interestingly, we can observe a slight improvement of the latency for our approach in comparison to the nic_bridge.

Since the TCP_RR benchmark does not measure the TCP connection setup, we accounted the additional latency imposed by the policy decision module separately. This overhead only occurs once for every TCP connection and is thus amortized over the lifetime of the connection. For this, we measured a maximum of 30 milliseconds. Note that our implementation is still in a proof-of-concept stage so that various optimization techniques could be applied to improve the performance.

VI. Conclusion

From the microkernel-perspective, using a dedicated network stack for each application is a common and reasonable design decision in order to achieve a high level of isolation. However, this approach either enables full and uncontrolled network access to potentially malicious applications or complicates the process of controlling the network accesses on a rather low abstraction level. In the scope of this work, we presented and implemented an alternative approach that consists in providing a single communication module that efficiently mediates and controls all network accesses. By deploying this communication module in a distributed system like an automotive system that feature rather complex networks of many ECUs, we can equip those systems with a distributed firewall that enforces the integrity of all network communication. As this communication module authorizes both, incoming and outgoing, connection requests, by invoking a policy engine, it protects the network from malicious processes on the ECU and the ECU from unauthorized network connections.

Nevertheless, as security often has a price, it is clear that our access-control mechanism imposes some overhead. For our approach, this consists in processing overhead by the communication module and protocol overhead required to provide secure network communication (i.e. IPsec). For the latter we have already shown in our preliminary work [21] that the overhead is typically very low. By introducing the communication module, we could marginally improve the average round-trip latency once a connection is established. However, this also requires some additional (but amortized) cost for establishing a connection.

In summary, communication integrity is an essential prerequisite for functional safety, a major requirement for automotive systems. We therefore believe that our approach enables the use of microkernels in such demanding distributed systems.

Acknowledgement

This work was supported by the DFG Research Unit Controlling Concurrent Change (CCC), funding number FOR 1800. We thank the members of CCC for their support.

References

[1] R. Charette, "This car runs on code," feb 2009. [Online]. Available: http://www.spectrum.ieee.org/feb09/7649

[2] A. Bouard, B. Glas, A. Jentzsch, A. Kiening, T. Kittel, F. Stadler, and B. Weyl, "Driving automotive middleware towards a secure IP-based future," in *10th conference for Embedded Security in Cars (Escar'12)*, Berlin, Germany, Nov. 2012.

[3] RTI Conntext DDS. [Online]. Available: http://www.rti.com

[4] M. Wolf, A. Weimerskirch, and C. Paar, "Security in automotive bus systems," in *Workshop on Embedded Security in Cars (ESCAR)*, 2004.

[5] Y. Laarouchi, Y. Deswarte, D. Powell, J. Arlat, and E. De Nadai, "Ensuring Safety and Security for Avionics: A Case Study," in *DAta Systems in Aerospace (DASIA)*, ser. ESA Special Publication, vol. 669, May 2009, p. 28.

[6] G. Heiser, "Secure embedded systems need microkernels," *USENIX ;login:*, vol. 30, no. 6, pp. 9–13, dec 2005.

[7] H. Härtig, "Security architectures revisited," in *10th ACM SIGOPS European Workshop*. New York, NY, USA: ACM, 2002, pp. 16–23.

[8] J. Liedtke, "Improving IPC by kernel design," *ACM SIGOPS Operating Systems Review*, vol. 27, no. 5, pp. 175–188, Dec. 1993.

[9] K. Elphinstone and G. Heiser, "From L3 to seL4 – what have we learnt in 20 years of L4 microkernels?" in *ACM Symposium on Operating Systems Principles*, Farmington, PA, USA, nov 2013, pp. 133–150.

[10] A. Lackorzynski and A. Warg, "Taming Subsystems: Capabilities As Universal Resource Access Control in L4," in *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems (IIES)*. New York, NY, USA: ACM, 2009, pp. 25–30.

[11] GenodeLabs. [Online]. Available: http://genode-labs.com

[12] Kernkonzept. [Online]. Available: http://www.kernkonzept.com

[13] Cog Systems: OKL4 Microvisor. [Online]. Available: http://cog.systems/products/okl4-microvisor.shtml

[14] QNX Neutrino RTOS. [Online]. Available: http://www.qnx.com/products/neutrino-rtos/neutrino-rtos.html

[15] T. von Eicken, A. Basu, V. Buch, and W. Vogels, "U-Net: A user-level network interface for parallel and distributed computing," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP)*. New York, NY, USA: ACM, 1995, pp. 40–53.

[16] R. Chutorash, "Firewall for vehicle communication bus," Feb. 24 2000, wO Patent App. PCT/US1999/017,852. [Online]. Available: http://www.google.de/patents/WO2000009363A1?cl=en

[17] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith, "Implementing a distributed firewall," in *Proceedings of the 7th ACM Conference on Computer and Communications Security (CCS)*. New York, NY, USA: ACM, 2000, pp. 190–199.

[18] V. Prevelakis and M. Hamad, "A policy-based communications architecture for vehicles," in *1st International Conference on Information Systems Security and Privacy (ICISSP)*, Feb. 2015, pp. 155–162.

[19] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis, "The keynote trust-management system version 2," RFC 2704, September 1999, http://www.rfc-editor.org/rfc/rfc2704.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2704.txt

[20] S. Kent and K. Seo, "Security architecture for the internet protocol," RFC 4301, December 2005.

[21] M. Hamad and V. Prevelakis, "Implementation and performance evaluation of embedded ipsec on microkernel os," in *The 2nd World Symposium On Computer Networks and Information Security*, September 2015.

[22] A. Garg and A. N. Reddy, "Mitigation of DoS attacks through QoS regulation," *Microprocessors and Microsystems*, vol. 28, no. 10, 2004.

[23] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. Mccoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage, "Experimental security analysis of a modern automobile," in *In Proceedings of IEEE Symposium on Security and Privacy in*, 2010.

[24] M. Unzner, "A split TCP/IP stack implementation for GNU/Linux," Diploma thesis, Technische Universität Dresden, 2014.

[25] Genode OS framework. [Online]. Available: https://genode.org/

[26] S. Carl-Mitchell and J. S. Quarterman, "Using arp to implement transparent subnet gateways," RFC 1027, October 1987, http://www.rfc-editor.org/rfc/rfc1027.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1027.txt

[27] (2014, feb) QNX Neutrino RTOS System Architecture. [Online]. Available: http://support7.qnx.com/download/download/26183/QNX_Neutrino_RTOS_System_Architecture.pdf

[28] (2014, mar) QNX Core Networking Stack User's Guide. [Online]. Available: http://support7.qnx.com/download/download/26171/Core_Networking_with_io-pkt_Users_Guide.pdf